A Generic Platform for Name Resolution in Source Code Analysis

Nicolas Anquetil, Yuriy Tymchuk, Anne Etien, Gustavo Santos, Stéphane Ducasse

A Generic Platform for Name Resolution in Source Code Analysis

Nicolas Anquetil[1], Yuriy Tymchuk[2], Anne Etien[1], Gustavo Santos[1], and Stéphane Ducasse[1]

[1] LIFL CNRS UMR 8022,
Inria Lille Nord Europe,
Université Lille 1, France.
{firstname.lastname}@inria.fr
[2] REVEAL – Faculty of Informatics, University of Lugano (USI)
Via G. Buffi 13, CH-6904 Lugano, Switzerland
yuriy.tymchuk@usi.ch

Abstract. Analysing a software system supposes two preliminary tasks: parsing the source code and resolving the names (identifiers) it contains. The parsing results in an Abstract Syntax Tree (AST) representing the source code. Name resolution maps all the identifiers found in the code to the software entities they refer to (variables, functions, classes, ...). If there are solutions for some popular programming languages (*e.g.,* JDT for the Java language), these two tasks can impose a significant burden on multi-language platforms (*e.g.,* Cast, Eclipse, Rascal, Spoofax, Synectique) where a parser with name resolution must be implemented for each language analysed. For the parser, one may use a grammar of the language and a parser generator tool. For name resolution, solutions are *ad-hoc* and one must develop them by hand. We work with a company that had to create parsers and name resolvers for five languages in the past 18 months. As a solution, we describe in this paper, an infrastructure that helps implementing a name resolution tool. This infrastructure is based on an AST metamodel similar to ASTM (from the OMG). One part of the solution comes from decomposing the task into two phases: First, looking-up for candidate entities that could map to a name; second selecting among these candidates the entity that actually maps to the name. Another part of the solution relies on the definition of scopes as first class entities that can be attached to any node in an AST. We discuss implementation of our solution for two languages: Ada and Pharo (a Smalltalk dialect).

1 Introduction

Modern IDEs provide functionalities such as code completion, identification of un-initialized variable, refactoring tools to rename an element or change its definition, etc. Many of these functionalities rely on name (or symbol) resolution to identifies all the uses of a given software component. Similarly, software quality

tools or software analysis tools are also based on the resolution of names in the source code analysed.

Providing such services for programs written in language not currently supported (old niche languages like Progress[3] or new ones as any Domain Specific Language) requires, as a first essential step, to develop a parser and then a name resolution algorithm for the language. The *parser* is the tool that will read a program in a given language (following its *grammar*), "understand" it, and transform it into an Abstract Syntax Tree (AST). The *name resolver* is the tool that will link all the uses of a name (*e.g.,* a variable name) in the AST to the definition of the software entity this name represents.

If the technology to perform both activities is old, it still represents a significant amount of work that can extend over weeks for full fledged programming languages. We are helping a start-up company that develops customized analysis tools. In the last 18 months, it had to create parsers and name resolvers for five programming languages. For the parsers, one can use parser generator tools to go from the grammar to the actual parser. But writing a name resolver is mostly a manual task that highly depends on the semantics of the analysed language.

In this paper, we propose a generic platform to ease name resolution. This solution stands on three legs:

- We adopt a model-based approach and use an AST metamodel similar to ASTM[4] from the OMG;
- We decompose name resolution in two steps, *lookup* and candidate *selection*;
- We define scoping as a first class entity that can be attached to any node of the AST.

For each symbol in the AST, the software entity to which it refers is looked for in the current scope and recursively in the scope(s) of its scope. This is the *lookup* phase which is mostly independent of the language. In a second phase, we *select* the appropriate entity in the list of candidate entities returned by the lookup. This part is more specific to the language, but generic patterns can be found.

Uses of this model-based generic platform are reported for Pharo [BCDL13] (a Smalltalk inspired language) and Ada.

The paper is organized as follows. In section 2, we describe the existing issues in name resolution. In Section 3 we describe the abstract syntax tree metamodel. In Section 4 we detail the generic name resolution algorithm. In Section 5, we adapt this algorithm for Ada and Pharo. In Section 6 we discuss related work and in Section 7 we conclude the paper.

2 Issues in Name Resolution

Name resolution amounts to linking a name (an identifier) in the source code to an entity of the program: in the expression `i++`, the symbol `i` presumably refers

[3] www.progress.com
[4] www.omg.org/spec/ASTM

to a variable of the program that must be incremented by one. In some cases, it might be possible to infer the kind of entity one is considering directly from the syntax of the source code. For example in the above expression, `i` should be a variable. In other case, the kind of entity is less clear. For example in the Ada expression `a(1)`, `a` could be a function or an array variable.

Finding to what entity the name refers, first depends on the scoping policy of the language: lexical or dynamic scoping. In *dynamic scoping*, the entity to which an identifier refers is looked for in the execution stack. This is used by languages as Dynamic Lisp or Perl. This can only be resolved at execution time and we will therefore not consider dynamic scoping in this paper.

In *lexical scoping* the name refers to the closest entity in the current lexical environment. Lexical environments are created by some language constructs like packages, classes, or functions. They are nested: a method scope is nested within its class scope, which is nested within its package scope. Most current languages use lexical scoping: C, Pascal, Java, etc.

One can add to this a "compile time scoping" for constructs like `#define` in C. These constructs are resolved at compile time according to the environment and options passed to the compiler.

The basic rule for name resolution in lexical scoping is to look for the entity in the current scope, e.g. a variable name will be first searched in the scope of the function within which it appears. If a matching entity is not found in the current scope, one searches recursively in the containing scope. But this generic algorithm has many variations according to the programming language. Here are some specificities of various languages:

- Some languages might not require to declare variables and they are created when first used. This does not really change the rules of name resolution. The variable is declared in the scope where it is first used.
- In Javascript a function can be used as a kind of class (rather a class constructor) by calling it with the `new` keyword. Also in Javascript, functions can be created without the `function` keyword, but by calling the `Function` constructor which is akin to creating a new instance of a Function class. Such a function can be called by the name of the variable in which the instance is stored (`var v = new Function("a", "b", "return a + b"); v(2, 6);`).
  These behaviours make it very tricky to apply name resolution within the function which is declared at run time.
- Languages can be case sensitive (Java, C, etc.) or not (Cobol, Pascal)[5]. This influences how names are matched, for example whether dog, Dog, and DOG all refer to the same entity or not.
- Same languages allow overloading functions or methods. This means two different entities (functions or methods) can have the same name. In this case, the matching also depends on the number and type of the parameters and the returned type (*i.e.,* the full signature).

[5] See: http://rosettacode.org/wiki/Case-sensitivity_of_identifiers, last consulted on 17/03/2014

- In Cobol, two entities of different kind (*e.g.,* a variable and a label) can have the same name. Thus name matching also depends on the kind of the entity.
- Some languages have instructions that creates a temporary scope. For example in Pascal the with statement places the statements it contains in the scope of a given structured type. Assuming there is a structured type `rec` containing an attribute `a` (*e.g.* "`type rec = record a:int; ... end;`") and a variable `v` of this type ("`var v:rec;`"). One normally accesses the attribute by writing `v.a`, but the `with` instruction creates a temporary scope such that within it, one can omit the "`v.`" prefix to access `v.a`. Thus "`v:rec; with v do writeln(a);`" will print the content of attribute `v.a`.
- In OO languages, on top of lexical inclusion of scopes (method scope included in the scope of its class), inheritance also defines an inclusion of scopes: the scope of a subclass is included in the scope of its superclass. If the subclass does not define a method, it must be looked for in the superclass definition. Thus a scope can have several parent scopes.
- OO languages also assume two implicit variables, `this` (or `self`) and `super`, that are never defined but accessible within the scope of a class.
- Some languages (*e.g.,* C++, C#, Java) have access modifiers (`public`, `private`, `protected`, and `default package`) that affect the scope of a definition.
- C++ has a `friend` access modifier that bypasses the `private` and `protected` modifiers (everything becomes `public` for a friend).
- Ruby, Objective-C, Pharo, C# offer package extensions. Packages can add methods to classes of other packages. For example package `p1` introduces class `C`, and package `p2` extends `C` with a new method `m` that was not originally defined. The extended method (`m`) is not available as long as the package (`p2`) that introduces it is not loaded. In practice, such an extended method can only be safely called from the package (`p2`) that introduces it.
- Finally, each language may have specific constraints on where some entities might be defined or not. In Pascal all scopes can declare functions (or procedures), types, and variables whereas in C a function can only be defined at the global scope. C and C++ allow defining variables at the global scope whereas it is not allowed in Java.

One can assume that every single programming language will have a set of specific constraints or rules that affects how name resolution works. It is clear from this list that a truly generic name resolution mechanism cannot be defined. There are too many rules depending on the language. Nevertheless, some characteristics are common to every languages such as the main idea of scopes nested into one another (a method scope nested in a class scope nested in a package scope) and searching for names up in this chain. Our generic resolution algorithm exploits these common points while allowing some adaptations. Before detailing the algorithm, we present the AST metamodel that enables a unified representation of programs written in various languages and is the base of our platform. Such AST Metamodel is similar to ASTM [AST11].

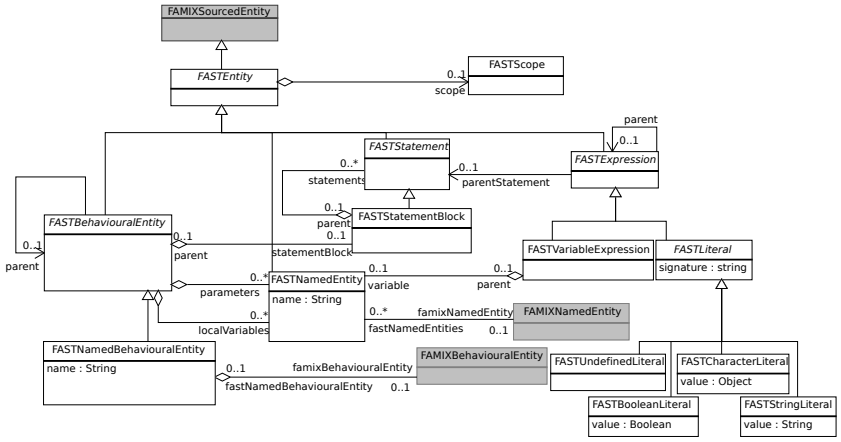

Fig. 1. The FAST metamodel (in white boxes), dark boxes represent concept from the Famix metamodel, used as symbol table.

3 The FAST Abstract Syntax Tree Metamodel

As it is common in name resolution, our algorithm relies on a representation of the program to analyse as an AST. This AST is a model of the program that follows the specifications of a metamodel (called FAST). This metamodel is important because it also constrains what the name resolution algorithm can do in a generic manner. For this reason, it is important to discuss our AST metamodel and its genericity.

Our metamodel has the same goals as ASTM [AST11], the standard defined by the OMG. However, we adopted a radically different approach in our definition. Whereas ASTM is defined with a focus on completeness, FAST focuses on genericity.

3.1 ASTM drawbacks

ASTM is composed of two parts: a core specification, the Generic Abstract Syntax Tree Metamodel (GASTM) and a set of complementary specifications that extend the core, called the Specialized Abstract Syntax Tree Metamodels (SASTM). GASTM defines a core set of modeling elements that are common to many programming languages. In fact, GASTM is the *union* of concepts from many languages. It considers object-oriented programming languages with concepts such as `ClassType`, `ExceptionType` or `AccessKind`. It has also concepts specific to procedural programming languages such as `JumpStatement` or `Pointer`. In total GASTM of the OMG defines 188 concepts. A part from this high number of concepts, the major drawback of this metamodel is the fuzzy semantic of the concepts. For example, if Java and Pharo[6], two OO languages,

[6] pharo-project.org

have packages, classes and methods, their definitions are different and so also the rules for scoping: in Pharo, classes can only be defined at package level. This suggests that having all the concepts in the metamodel might not prevent us from having to specialize them for each language.

3.2 FAST metamodel

Our metamodel, FAST, is defined as the *intersection* of all programming languages. By doing this, we have a metamodel with less than 20 concepts that can still accommodate the same large spectrum of programming languages while being much easier to apprehend and extend. FAST, just as GASTM, has to be specialized but the existing core is concise, generic and with the same semantic in each language.

The result is presented in Figure 1 (white boxes). It starts with an abstract concept `FASTEntity` serving as the root class of the FAST metamodel. A `FASTEntity` may have a scope `FASTScope` or not. Four types of entities are distinguished:

- A `FASTBehaviouralEntity` is an abstract concept for all entities having a behavior like methods or functions. Such entities may be named (in most cases) or not (*e.g.,* lambda-functions).
- A `FASTStatement` is also an abstract concept. *IfStatement* or *LoopStatement* do not exist in all languages. For example, they do not appear in Pharo or Cobol. These two languages do offer the possibility of branching and iterating, but not in the form of the "traditional" statements we know in Java, or C.
  Some languages may also offer specific loop statements (as the "extended for" in Java).
  A *ReturnStatement* is probably universal in languages that accept subprograms, but it will not always return a value (*e.g.,* Cobol). Similarly, an assignment may be considered as an expression-statements in many languages (meaning it returns a value), whereas it is a simple statement in other (*e.g.,* Pascal or Cobol). In the end we chose to be conservative and did not include any specific statement in the core FAST.
  Statements can be `FASTStatementBlock`, for example to represent the body of a function.
- A `FASTExpression` is an abstract concept that has a value. Again it would be difficult to try to be too specific here as even arithmetic expressions can be treated in different ways by different languages (*e.g.,* Pharo, Lisp). We believe some literals (`FASTLiteral`) are truly generic and included them.
- A `FASTNamedEntity` represents an identifier. Most of the type it maps to variables or types.

All these concepts are very generic and exist in any language. They don't capture the specificity of any language or even any paradigm (procedural, object, functional, ...). Our name resolution algorithm as shown in the Section 4 relies only on these concepts making it generic.

3.3 FAST and Famix Interconnection

FAST is an extension of the Famix metamodel [DTD01,DAB+11]. FAST actually stands for Famix AST metamodel and relies on the Famix concepts to represent structural part of the code (packages containing classes, containing methods).

Famix offers a structural representation of the source code. It has packages, classes, methods, functions, variables (parameters, attributes, local variables, etc.). Typically in Famix, one also stores relationship between entities (invocations between functions, accesses to variables, etc.) but we are not using this part here as it already requires name resolution. So we work with a simplified Famix model. The only relationship we require between the Famix entities are the structural ones (parent or container), which do not depend on name resolution but are obtained from traversing the AST top-down.

A `FAMIXSourcedEntity` models any fact in a source program and it is the superclass (root class) of all source code entities and their relationships. The `FAMIXNamedEntity`, `FAMIXBehaviouralEntity` and `FAMIXSourceEntity` concepts (in light grey in the figure) are the points where we anchor FAST to FAMIX has shown in the figure.

We also use Famix as the "symbol table" of our algorithm. This means that names found in the AST (in a FAST model) are linked to entities defined in the Famix model.

4 Generic Name Resolution

We need name resolution for program analysis purposes. In consequence, we assume the analysed programs are valid. This means, we assume a name does resolve to some entity, we do not check for errors. This does not seem to be a significant simplification of the problem as an error would simply mean that there is no resolution for a name.

4.1 Generic Name Resolution Algorithm

The generic algorithm is decomposed in two parts: *lookup* and *selection* as illustrated in Figure 2. The *lookup* is the most generic part. From a name in an AST, it generates an ordered list of candidate entities that it could resolve to. Given this list of candidates, and the referring entity, the *selection* finds the first candidate that answers all the criteria of the programming language. This part is more language dependent as explained in Section 2, however, generic patterns can be found.

4.2 Scopes as first class entities

Before going to the algorithm itself, we must discuss the notion of scopes. A scope is a partition of the system in subsets. In a scope, a name always refers to the same entity whereas the same name may refer to different entities if it

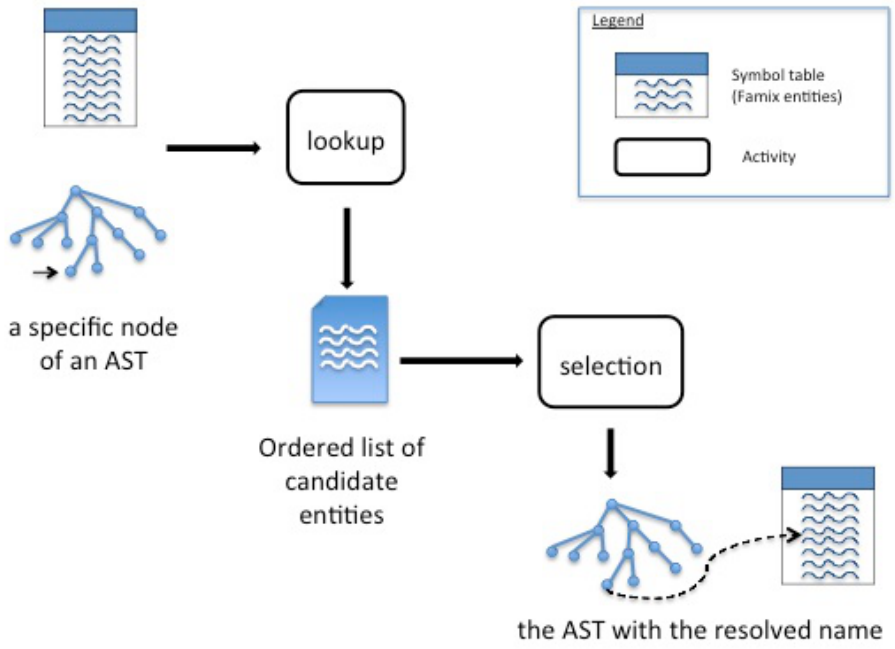

Fig. 2. Sketch of the algorithm

is encountered in different scopes. Therefore a scope is in essence a dictionary that associates a name to an entity. Sometimes, the kind of entity must also be taken into account (variable, function, type, ...), in [KKWV13] this is treated by creating different namespace for each entity kind. We delegate this to another part of the discussion (Section 4.4).

Traditionally scopes are attached to specific constructs of the language. Thus in Java one may speak of the scope of a package, the scope of a class, or the scope of a block. For more flexibility, we consider the scopes as independent entities that can be attached to any FAST node (and even to several).

Scopes form a containment chain. For example, the scope of a `FASTStatement-Block` may have for parent the scope of a method definition, `FASTBehavioural-Entity`, that would have in turn for parent the scope of a class definition. Scopes can have more than one parent. Scopes of OO classes usually have for parent the scope of their containing package plus the scopes of each superclass. These parents are ordered, in Java, the first parent would be the scope of the superclass, then the scopes of the implemented interfaces in the order of the `implements` declarations, and finally the scope of the parent entity (typically a package). When looking for a name, the algorithm will look in the parent scopes in this order, first the superclass, ..., and the parent package in the end. The rational is that, if *aVariableName* is an attribute of a class, even if it is inherited, it must have priority over a global variable with the same name defined in the package

of the class. We believe this is a generic rule although we did not check all OO languages[7].

Not all AST nodes have a scope, statements and expressions often do not, but all can have one, for example the `for` statement in Java and C may introduce new variables "`for (int i=0;...`" so it will have a scope attached. In Pascal the same `for` statement does not have an attached scope, the variable must be defined in the enclosing function. Knowing which AST node has a scope or not is language dependent and decided by the parser of that language.

Being able to attach a scope to an arbitrary node allows us to model many features. Thus a Java import can be implemented by attaching the scope of the imported package to the import statement (see Figure 3). The Java class declared in the importing file will then have a scope whose parent will be the scope of the imported package. This way, when looking for a name, it will be searched first in the scope of the class and if not found, the search will go up the "containment" chain of scopes and look in the scope of the imported package.

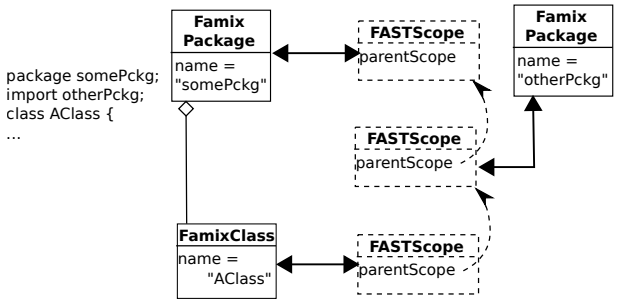

Fig. 3. A package import in Java. The parent for the class' scope is attached to the imported package to allow resolving the name of imported classes from this scope.

Accesses to structured variable's fields (or class instance members) can also be represented by reusing scopes. We attach to a structured variable (procedural languages) or variables containing instances of a class (OO languages), the scope of their type (see Figure 4). That way, when using the dot notation (`v.a`), the member will be looked in the scope attached to the variable which is that of its type. Similarly a Pascal `with` statement (described in Section 2), will have attached to it the scope(s) of the record type(s) of the variable(s).

4.3 Generic Lookup Algorithm

The *lookup* for a name starts from the AST node where that name appears and goes up the AST to find the first node with a scope (*e.g.* a statement-block

[7] It does matches Java, C++ and C# behaviors

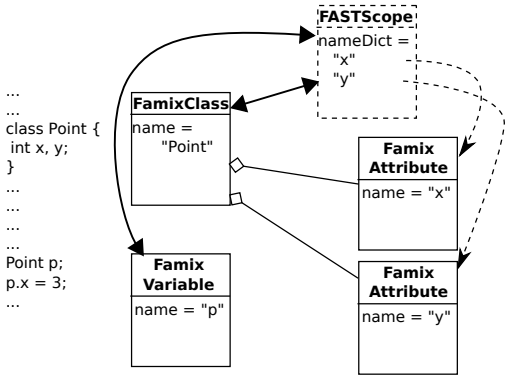

Fig. 4. A class with two attributes (up left and down right), and an instance (down left). The class' scope (up right) is also attached to the variable to allow resolving the name of the members (`p.x` here).

between curly brackets in Java, a function, or a class). It searches for entities with the given name in this first scope. Independent of the result of this search, it then recursively searches more candidate entities in the containing scopes, up to the top-most scope ("universe" scope). Candidates are returned in order of proximity; candidates in the immediate scope appear before candidates from a parent scope. This rule is one of the foundations of lexical scoping and is therefore generic. The output of this first part is an ordered list of candidate entities.

Testing the match between an entity and the AST node containing a name is delegated to this AST node. For the name this is because some languages are not case sensitive (*e.g.,* Pascal) whereas others are. In this case, all node for a given language will react the same way and use the appropriate name matching test. Case-sensitive and case-insensitive tests are implemented in the infrastructure and, typically, all nodes in an AST access the same test. For the entity kind, each node must know what it can accept and this is part of the definition of the AST nodes. For example, a `FASTMessageSend` contains the name of the method called (in `selector`, see Figure 6). This node will only accept as candidate method entities. On the other hand, a `FASTNamedEntity` will typically accept variable entities or type entities. Note however, that matching the kind of entity could be delegated to the selection phase. This is an alternative option that we are still considering.

The nodes of core FAST, are provided with default behaviors. Languages requiring specific behavior will subclass these core nodes. In this sense, this part of the algorithm is as generic as it can be, by delegating a part of the work to the AST nodes which are created by the parser for the language.

4.4 Generic Selection Algorithm

Given an ordered list of candidate entities, the *selection* algorithm will return the first candidate that matches the rules of the language. This part is dependent of the language but may be made more generic by the use of different selectors, for example to deal with the cases of `public`, `protected`, or `private` entities.

Each element of the ordered list of candidate entities is successively studied. By construction of the lookup algorithm (see previous section), it matches the searched name and the expected kind.

The selection consists in checking if the software component that uses the name may access to each candidate entity or not. For this it needs to know the accessing entity and the candidate accessed entity. From this, it will usually look at the visibility of the candidate entity, and where both entities (candidate and accessing) are declared. Each non accessible entity is eliminated from the list of candidates. The first candidate satisfying the accessibility conditions of the language is returned by the name resolution algorithm.

So, the selection phase first looks for the accessing entity. This is simply done by going up the AST to find the innermost node that corresponds to an entity definition (such as a `FASTBehaviouralEntity` node, corresponding to the definition of a method or a function). Again knowing which AST node corresponds to an entity definition is language dependent, but there are very few of them (typically, classes/types and methods/functions).

Then the selection phase must find the first candidates entity verifying the access rules of the language. This part is the most language dependent, for example in C++, it will also depend on the access modifiers of the candidate entities as well as possible declaration of `friend` classes. To help in implementing this part, our infrastructure proposes several selectors to check for the basic access modifiers: `public`, `protected`, `private`, `default-package`.

- For public entities, the `FASTPublicSelector` should be used. It accepts all candidates. As a candidate returned by the lookup, the candidate entity already matches name and kind; being public, it is accessible by all entities. Note that, older languages (C, Pascal, Ada, Cobol) typically only consider `public` definitions.
- For private entities, the `FASTPrivateSelector` should be used. It refuses all candidates except if accessing entity is in the same class as the candidate entity.
- The `FASTProtectedSelector` checks the accessibility rule when the access modifier is `protected`. It verifies that the accessing entity is in a subclass of the class owning the candidate entity, if not it eliminates the candidate.
- The `FASTDefaultPackageSelector` implements the default package accessibility rule. It verifies that the accessing entity is in the same package as the candidate entity.

4.5 Illustrating Example

We now explain on a Java example, how the generic algorithm works (see Figure 5). We consider a superclass `ClassA` that defines an inner class `InA`, itself with

an attribute `att`. We also have `SubA`, a subclass of `ClassA`, with a method `m()`. This method defines a local variable `v` of type `InA` and accesses the attribute `v.att`.

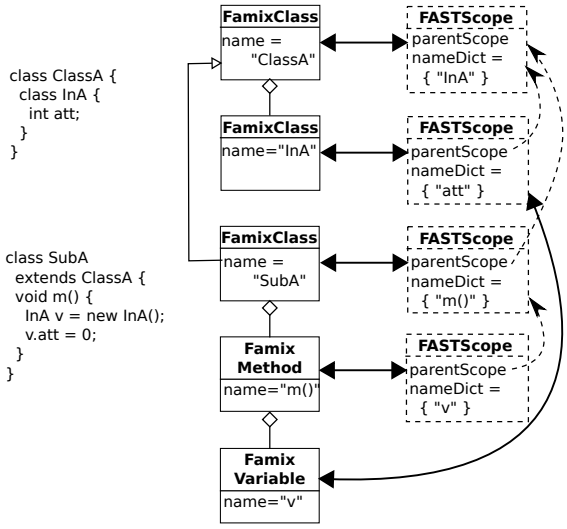

Fig. 5. Code sample for Java name resolution and the associated FAST scopes. For the sake of space, the actual AST is not shown.

First, some name resolution occurs at the beginning of method `m()` for the name "InA" (the type of `v`). The lookup algorithm searches for it in the enclosing scope, that of the method body (lowest scope). It finds nothing. Enclosing scopes are then looked-up: that of the class `SubA`, the superclass `ClassA`, the package of `ClassA` (not show in the figure), and finally the package of class `SubA` (not shown either). In `ClassA`, a candidate entity is found, matching both name and kind of "InA". There is no other entity "InA" defined in any of the other scopes. The selection algorithm tests this candidare using `FASTDefaultPackageSelector` to verify that `m()` can access this `InA`. The result is positive the candidate is returned by the name resolution mechanism.

From this, the scope of class `InA` is attached to the variable `v`.

Then on the last statement of `m()`, the name "v" must be resolved. First the lookup algorithm goes up the AST to find a node with an associated scope. It finds the scope for method `m()`. So it starts searching for the name "v" in this scope and its parents, the scope of `SubA`, etc. Lookup finds only one candidate that is accepted in the selection phase.

Finally, the name "att" is resolved, first in the scope attached to the variable and then up the containment chain of scopes. In the scope for class The selection

algorithm will first look at the `InA.att ClassA` the selection algorithm finds an entity mathcing the name, and this one is selected by the selection algorithm.

5 Practical Application

We now explain how FAST and the generic algorithm for name resolution can be applied to two different languages: Ada, a procedural language, and Pharo, an OO one. Applying to a new language actually means:

- Specializing the FAST core metamodel by adding elements necessary to represent the concepts of this language;
- Writing a parser for the language and generating the AST;
- Deciding which FAST elements specific to the language have a scope, and;
- Implementing a selection strategy eventually reusing the existing selectors.

5.1 Pharo (Smalltalk)

We first dealt with Pharo, a Smalltalk inspired language [BCDL13]. It is an object-oriented language that as the advantage of having a very simple AST metamodel that we can fully represent here (Figure 6). It is also interesting as it introduce the notion of blocks that is rarely encountered in other languages and is interesting in the context of this paper.

Specializing the AST. One specificity of Pharo is that it has a extremely simple grammar with very little statement or expression nodes. Almost everything is done by sending messages: class and object creation, iterations and tests, expressions, etc.

Has a result, we could create a full AST metamodel with only 13 new concepts. They respectively extend `FASTStatement`, `FASTExpression` and `FASTLiteral`. For example, `FASTReturnStatement`, `FASTExpressionStatement`, several types of expression and literals have been added. `FASTMessageSend` is an important concept as previously explained.

We introduced the notion of block definition with `FASTBlockDefinition`. In Pharo, a "block" is a closure (similar to lambda-functions). Blocks may have parameters, define local variables, and contain statements.

Figure 6 presents the resulting metamodel where the dark grey nodes are the core FAST one (generic, language independent, AST from Figure 1), and white nodes for the Pharo extensions.

Determining elements with scope. In Pharo, classes may be defined in packages, but these ones are not namespaces, only organizational units. This means that even in different packages, two classes cannot have the same name, they can be considered as public.

In other words, packages do not have `FASTScope` and classes are defined globally.

Classes define a lexical scope, several classes can have methods with the same name. Methods also define a lexical scope, they may have local variables. Blocks

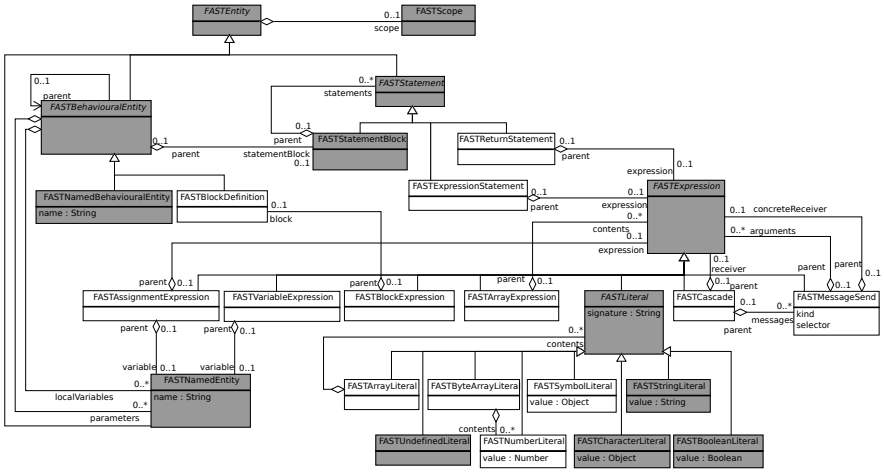

Fig. 6. Pharo AST metamodel in FAST (white boxes), the dark boxes are concepts from the core FAST metamodel (see Figure 1)

can define local variables or other blocks. They are similar to methods and have a scope. Finally, there is a global scope (the Pharo environment) that may contain classes and global variables

Accessibility definition. Classes and methods are accessible from everywhere (they are "public"). Attributes are not accessible outside of their class (they are "protected").

The selection algorithm can be implemented by using the platform existing selectors (`FASTPublicSelector` and `FASTProtectedSelector`) and calling the appropriate one depending on the kind of entity.

5.2 Ada-95

Ada is a procedural language with an extensive grammar. We were not interested in a full AST, but focused on procedure and function calls. This will allow us to illustrate how one could use our platform to work with island grammars, *i.e.* grammars that specify only a part of the language, leaving other parts as an undefined list of characters or tokens[8].

Ada is a case-insensitive language, for this we used the tool that converts all names to uppercase before doing the comparison (in the lookup part).

Specializing the AST. To extract procedures, functions and their invocations, we first need to recognize their declarations, and that of their parent scopes. We thus created FAST nodes for packages, tasks, and subprograms (procedures and functions).

[8] Note however that in this case we did have a full Ada-95 parser, but only extracted a small subset of all AST nodes

We represented the `with` keyword that is somehow similar to an import in Java, it specifies that a source file may use the definitions of another package[9].

We also needed to recognize the invocations of subprograms. Procedure invocations are recognize directly in the grammar and we created a `FASTAdaProcedureCall` as a sub-concept of a `FASTStatement`. As mentioned in Section 2, functions invocations are syntactically similar to other construction as array indexing. This means a function invocation `f(x)` can only be identified by resolving the name `f` to a function definition. Therefore, we had to specify the part of the AST for Ada expressions that is related to `QualifiedIdentifier`. Fortunately, this excludes all arithmetic and logical expressions which are a large part of a grammar. We defined `FASTAdaQualifierKeyword` as a subconcept of `FASTExpression`, and a `FASTAdaArgumentsSelector` to represent the "`(x)`" part.

Determining elements with scope. The nodes with scopes are the packages, tasks, and subprograms. We dealed with the `with` clause by attaching a scope to the `FASTAdaWithClause` as explained in Section 4.2, Figure 3.

Accessibility definition. Ada has a `private` access limitation, but it applies only to types, therefore it did not concern us. All elements of interest to us are public.

6 Related Work

In [KKWV13], the authors identify recurring patterns for name binding and introduce a metalanguage to specify name binding in terms of namespaces, scopes. They provide a language parametric algorithm for static name resolution during compile time. Their approach differs from ours in a number of ways.

First they aim at making the name resolution rules of a language explicit through a DSL they describe. Our goal was primarily to have a generic name resolution solution and we opted for a more programmatic "description" of the rules. However, by decomposing the overall name resolution task in two subtasks and identifying further subparts of these tasks (*e.g.,* the various selectors described in Section 4.4) we made some steps in the direction of having the name resolution rules of a languages being described at a higher level of abstraction than just raw source code.

Second, interestingly Konat *et al.* also decompose their solution into three subtasks that are slightly different from ours. the first phase all definitions are mapped to an entity, we do not consider this phase, as it is very straightforward and independent of the language. For us definitions are mapped to entities defined in our Famix metamodel. In the second phase, they do type inference, something that we explicitly left outside of the scope of this paper (see Section 4.5). What we describe in this paper seems, therefore, to correspond to their third phase. We have decomposed this into two more detailed steps.

Third, they have different usages than ours. Their solution is integrated into an IDE, with code edition, code highlighting, compilation error checking, refac-

[9] In this case, a company rule prohibits to use the `use` keyword for other things than types, so we did not need to consider it.

toring, etc. Among other things, this implies that they must deal with incomplete and erroneous programs. As mentioned in Section 4, for now, we are assuming a complete and valid program (that compiles). It is not clear to us at this point whether this is a significant restriction (whether it makes any difference). This is something we did not test.

Finally, although they claim language independence, all the examples given in the paper are focusing C++. Actually dealing with different languages does imply some amount of tweaking.

In [BPM04], the authors propose the DMS Software Reengineering Toolkit, a generalized compiler technology. Their approach and ours share the same purpose: providing a generic name resolution system. However, DMS relies on a representation of the AST as a hyper graph and not as a model as in our approach. Furthermore, as far as we understood from the paper, the look up function is a parameter of their algorithm that the developer must provide. By decomposing the algorithm into lookup + selection, we can reuse more parts and we expected that our look up algorithm should be already generic enough for new languages, and our selection algorithm would need to be extended in very few language instances.

In [KRV10], the authors adopt a textual modeling approach and propose a framework named MontiCore for the compositional development of textual DSL and their supporting rules. Concrete and abstract syntaxes are defined using the MontiCore grammar. They provide default implementations for simple resolving problems like file-wide flat or simple hierarchical namespaces. Similarly to [BPM04], more complex resolution algorithms are let to the responsibility of the programmers. In [JBK06], the authors propose a similar approach based on a DSL named Textual Concrete Syntax to provide a concrete syntax for an abstract syntax given as a metamodel. So contrarily to [KRV10], abstract and concrete syntaxes are defined in two different languages. Concerning the name resolution algorithm, only simple cases are tackled: unique symbol tables or nested ones. The way the tables are nested is not described in the paper.

The OMG has defined two metamodels to specify concrete and abstract syntax, KDM (Knowledge Discovery Metamodel) [PCdGP11] and ASTM [AST11] respectively. KDM specifies a set of common concepts required for understanding existing software systems, whatever the used language, in preparation for software assurance and modernization. ASTM has been previously introduced in section 3.1. It is divided into two parts GASTM that involves syntactical concepts that are common in different programming languages and SASTM that extend the first to represent specificities of languages. The combination of those standards provides a modelling framework for designing and analysing software syntax and semantics. The purpose of these two metamodels and FAST are the same: providing a core metamodel to represent concepts common to different programming languages. Nevertheless, FAST core is reduce to the strict minimal set of concepts and gives more place to specific extensions. Indeed some concepts (like method or function) may have the same name in several languages but be a little bit different and thus appear only in the extensions. Moreover, no

name resolution algorithm is provided by the OMG or other authors on these metamodels.

7 Conclusion

Name resolution is a fundamental part of most language parsing activities, whether it is for compiling a program, or analyzing it. It is needed if one wants to refactor, build a call graph, analyze module dependences, etc. When defining parsers for various programming languages in our Moose platform [DTD01] we often had to face the task.

In this paper, we propose a generic name resolution algorithm based on FAST, an AST metamodel. The algorithm is composed of two parts, the *lookup* that searches for candidates entities matching a name in a chain of parent scopes from the point where this name is used; and the *selection* that chooses the entity responding to access rule specific to each language. By the definition of several selectors, depending on the visibility modifier of each candidate entity, we are able to answer the need of many different languages.

Future works include checking the genericity of this solution in more languages (*e.g.,* Python is starting) and more paradigms (*e.g.,* Lisp). We also expect to extend a bit some core functionalities. The main remaining work is to try to make more generic the selection part of the algorithm.

References

[AST11] *Architecture-driven Modernization: Abstract Syntax Tree Metamodel (ASTM)- Version 1.0.* Object Management Group, January 2011.

[BCDL13] Alexandre Bergel, Damien Cassou, Stéphane Ducasse, and Jannik Laval. *Deep Into Pharo.* Square Bracket Associates, 2013.

[BPM04] Ira D. Baxter, Christopher Pidgeon, and Michael Mehlich. Dms®: Program transformations for practical scalable software evolution. In Anthony Finkelstein, Jacky Estublier, and David S. Rosenblum, editors, *ICSE*, pages 625–634. IEEE Computer Society, 2004.

[Cob94] IBM. *COBOL/400 Reference — IBM*, 1st edition, June 1994. Available at http://publib.boulder.ibm.com/iseries/v5r2 /ic2924/books/c0918130.pdf (last accessed on 10/01/2013).

[DAB+11] Stéphane Ducasse, Nicolas Anquetil, Usman Bhatti, Andre Cavalcante Hora, Jannik Laval, and Tudor Girba. MSE and FAMIX 3.0: an interexchange format and source code model family. Technical report, RMod – INRIA Lille-Nord Europe, 2011.

[Dol05] Julian Dolby. Using static analysis for ide's for dynamic languages. In *The Eclipse Languages Symposium*, 2005.

[DTD01] Serge Demeyer, Sander Tichelaar, and Stéphane Ducasse. FAMIX 2.1 — The FAMOOS Information Exchange Model. Technical report, University of Bern, 2001.

[JBK06] F. Jouault, J. Bézivin, and I. Kurtev. Tcs:a dsl for the specification of textual concrete syntaxes in model engineering. In *Proceedings of the 5th international conference on Generative programming and component engineering*, pages 249–254, New York, 2006. ACM Press.

[KKWV13] Gabriël Konat, Lennart Kats, Guido Wachsmuth, and Eelco Visser. Declarative name binding and scope rules. In *Software Language Engineering*, pages 311–331. Springer, 2013.

[KRV10] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Monticore: a framework for compositional development of domain specific languages. *International Journal on Software Tools for Technology Transfer*, 12(5):353–372, 2010.

[LTP04] Timothy Lethbridge, Sander Tichelaar, and Erhard Plödereder. The dagstuhl middle metamodel: A schema for reverse engineering. In *Electronic Notes in Theoretical Computer Science*, volume 94, pages 7–18, 2004.

[PCdGP11] Ricardo Pérez-Castillo, Ignacio García-Rodríguez de Guzmán, and Mario Piattini. Knowledge discovery metamodel-iso/iec 19506: A standard to modernize legacy systems. *Comput. Stand. Interfaces*, 33(6):519–532, November 2011.